Impact-minimizing Runtime Switching of Distributed Stream Processing Algorithms

Cui Qin
Institute of Computer Science
University of Hildesheim
Universitätsplatz 1
D-31141 Hildesheim, Germany
qin@sse.uni-hildesheim.de

Holger Eichelberger
Institute of Computer Science
University of Hildesheim
Universitätsplatz 1
D-31141 Hildesheim, Germany
eichelberger@sse.uni-hildesheim.de

ABSTRACT

Stream processing is a popular paradigm to process huge amounts of data. During processing, the actual characteristics of the analyzed data streams may vary, e.g., in terms of volume or velocity. To provide a steady quality of the analysis results, runtime adaptation of the data processing is desirable. While several techniques for changing data stream processing at runtime do exist, one specific challenge is to minimize the impact of runtime adaptation on the data processing, in particular for real-time data analytics.
In this paper, we focus on the runtime switching among alternative distributed algorithms as a means for adapting complex data stream processing tasks. We present an approach, which combines stream re-routing with buffering and stream synchronization to reduce the impact on the data streams. Finally, we analyze and discuss our approach in terms of a quantitative evaluation.

Keywords

Data stream processing; runtime adaptation; impact-minimizing adaptation enactment; algorithm switching

1. INTRODUCTION

Big data applications aim at processing huge or complex data sets, which usually cannot be handled by traditional approaches. Distributed stream processing [2], i.e., continuous processing of conceptually endless streams of data items, is a popular approach to realize Big data applications. Depending on the actual application area, the stream characteristics such as volume or volatility can vary over time. For example, in the financial domain hectic markets can cause bursty streams leading to changes of the stream characteristics of several orders of magnitude. To cope with such situations, adaptation of the data processing at runtime is desirable.

While approaches such as Borealis [3] or RTSTREAM [17] provide adaptation capabilities for continuous stream queries, such as changing the query program, they concentrate on a fixed set of database-like stream operators. In contrast, recent frameworks such as Apache Storm[1] or Spark[2] support arbitrary analysis algorithms, but currently do not provide much support for runtime adaptation. Such arbitrary algorithms provide more freedom regarding the data analysis tasks to be realized and, in particular, can themselves be distributed, i.e., realize complex analysis tasks such as financial correlation computations in a scalable manner.

One specific way of adapting stream processing is to switch among different algorithms, which provide similar functionality but operate at different runtime characteristics [5]. This enables us to oportunistically utilize the better algorithm, e.g., at high load a faster, but more expensive algorithm such as a hardware co-processor, which can be utilized at low load to other more urgent analyses. However, a naive approach to switching can cause disturbances in the data streams and, thus, affect subsequent processing or lead to accidentally ongoing processing after the switch.

While adaptation for algorithms running on single nodes can be achieved with existing techniques, the research question in this paper is *How to realize an impact-minimizing runtime switching among alternative distributed (real-time) data stream processing algorithms?* At a glance, switching an algorithm at runtime may appear rather similar to adapting the query plan in known approaches, but, in contrast, we aim at arbitrary, distributed, potentially stateful data analysis algorithms. Our contribution is an approach for runtime switching among distributed data analysis algorithms, which aims at minimizing the impact on the data streams that adaptation can cause, such as missing, repeated, disordered or (massively) delayed data items. Therefore, our approach considers the memory state of the involved algorithms and synchronizes the (common) output stream. We show that our approach can perform a runtime switch in 60-110 ms depending on the setting and that the output data stream is not seriously affected. In this paper, we focus on switching techniques and, therefore, making the actual decision for adapting the data processing is out of the scope and subject to future work.

The work we presented in this paper has been performed in the EU-funded project QualiMaster[3], which aims at building a real-time adaptive data processing infrastructure. For demonstration, the project creates algorithms and applica-

BDPR '15 March 15, 2015, Bordeaux, FR

[1]http://storm.apache.org/
[2]http://spark.apache.org/
[3]http://qualimaster.eu

tions for analyzing systemic market risks. In this context, we apply our approach to enable the setting described above, i.e., to dynamically switch from a software-based execution to a hardware co-processor wrapped by a distributed software algorithm, e.g., to cope with dynamic load changes.

The remainder of this paper is structured as follows: In Section 2, we discuss related work. In Section 3, we introduce our approach in terms of two variants, a simple one as basis for later analysis and an advanced runtime switching including state maintenance and stream synchronization. We evaluate both variants empirically in Section 4. Finally, in Section 5 we conclude and provide outlook on future work.

2. RELATED WORK

In this section we discuss related work in terms of adaptive stream processing engines and mechanisms for adapting the processing. For minimizing the impact, it is important that the target algorithm takes over the processing as soon as possible and that the produced output stream does not have duplicated or missing items (in our context even the sequence of output items is not changed). Therefore, we also discuss work on migrating the memory state of algorithms at runtime and on synchronizing (output) data streams.

Several **adaptive stream processing engines** are described in literature, for example Borealis [3], RTSTREAM [17], CAPE [11, 13] or BiStream [10]. Typically, these approaches focus on continuous queries over data streams expressed in terms of a query plan consisting of (fixed) database-like operators such as project or join. In contrast, recent stream processing frameworks such as Storm or Heron [9] allow the data analyst to implement own operators (we call *algorithms*). However, currently these frameworks mostly focus on static processing, e.g., called "topology" in Storm, and do not provide much support for adaptive processing.

Different **mechanisms to adapt stream processing** are proposed in the literature. We now provide an overview of the most closely related mechanisms. Data admission (also called load shedding) [3, 17] is frequently applied to handle overload situations. Scaling along the compute resources is a further mechanism to counter high and varying load. Among others, Kulkarni et al. [8] adapt thread priorities, Lin et al. [10] adapt the amount of processing servers or the authors of [11, 14] migrate operators among servers to balance the load. Further, the structure of the data processing can be adapted without changing its semantics. E.g., the authors of [11, 13] dynamically rewrite the query plan by running the new plan (or changed parts) in parallel until the new plan can safely take over (called *parallel track strategy*) [13] or to re-route data streams [4]. Another option is to make the streams adaptive, e.g., in terms of their transfer batch size [14, 16] or their implementing parallel connections [16]. Probably, the closest approach to ours is by Hwang et al. [6], who switch among active and standby version of the same operator to recover from processing errors, but not among alternative algorithms. Although several mechanisms are available to adapt data stream processing at runtime, the related evaluations mostly do not take the impact on the data streams into account. Among the work cited above, the overall effect on throughput and latency is only evaluated in [10, 16], while Wei at al. [17] also measure data completeness (10 seconds processing disruption at 25k input rate) and Hwang et al. [6] achieve a recovery time of 50-170ms. Although there is similar work, we focus on switching among distributed user-defined algorithms, while minimizing the impact on the processed data streams.

When switching among stateful algorithms, it is important to enable the target algorithm to take over processing at a certain point in time. Therefore, the **memory state** must either be built up or it must already be up-to-date. One type of strategies is based on running operators or new query plans on demand in parallel to build up the required state, e.g, the parallel track strategy in [13]. Hwang et al. [6] describe an expensive strategy to perform parallel processing on standby operators. Another type is based on transferring memory among similar stateful stream operators [13], in particular transferring smaller partitions can be rather efficient [11, 14]. To avoid state transfer, Wu et al. [18] rely on a state sharing mechanism while changing the parallelization of stateful operators. Moreover, Aly et al. [1] incrementally complete missing states along the query plan. However, these strategies are typically only effective if there is a common memory state. In contrast, our work deals with arbitrary user-defined algorithms, which can even include algorithms for hardware co-processors optimized by different memory structures. Thus, we currently rely on a parallel track strategy for warming-up alternative algorithms.

When the actual runtime switch happens, in a short time frame data items may be duplicated or items may even be missing without further consideration. This can be mitigated by explicitly **synchronizing the output stream**. Some synchronization approaches are discussed in literature, typically relying on some notion of a synchronized (wall) time. For example, in media stream processing, Rothermel et al. [12] calculate the target media time for synchronizing distributed media streams from the synchronized time in the compute cluster. For continuous query processing, Ji et al. [7] buffer and sort disordered items according to their timestamps to achieve a higher result accuracy. However, in our context a sequential item identifier is more appropriate as the alternative algorithms typically process data at different speed and so timestamps are usually misleading.

In summary, to our very best knowledge there are no approaches on dynamically switching among arbitrary data stream processing algorithms, which explicitly aim at minimizing the impact on the involved data streams.

3. DYNAMIC ALGORITHM SWITCHING

In this section, we describe our approach to the runtime switching among distributed data stream analysis algorithms at low impact. More specifically, we introduce first some terms and discuss requirements characterizing the potential impact on the data streams. Then, we present two approaches: In Section 3.1, we discuss simple algorithm switching, which we use as a basis for the evaluation in Section 4. In Section 3.2 we present an advanced approach considering memory state and stream synchronization.

Dynamic algorithm switching performs at runtime a switch among *alternative algorithms*, e.g., from an algorithm family of functional similar algorithms having different runtime trade-offs [5]. We call the currently running algorithm *active*, the remaining alternative algorithms *passive* and the passive algorithm that shall be enabled by the runtime switching the *target algorithm*. Our goal is to minimize the impact on the characteristics of the processed data streams when switching data stream algorithms at runtime. We operationalize this goal in terms of four requirements, actually a refinement of the "process and respond instantaneously" requirement from the real-time stream processing requirements by Stonebraker et al. [15]:

R1. **No missing or duplicated data:** Switching an algorithm at runtime must not cause data loss or duplication of data items.

R2. **Transparency:** Although stream processing systems shall be resilient against stream imperfections [15], in our project adaptation mechanisms shall maintain the item sequence, i.e., be transparent in this regard. Thus, as a refinement of R1, an approach must not disturb the item sequence produced by the active algorithm, in particular not while switching.

R3. **Minimizing the switching time:** Switching algorithms at runtime shall happen as fast as possible to reduce the time for causing disturbances to the streams.

R4. **Minimizing effects on stream characteristics:** In addition to R3, also the effect on further (application-) relevant stream characteristics such as latency, throughput or volatility shall be minimized.

In this paper, we focus on throughput as one particular stream characteristic (R4). Other characteristics as well as non-functional aspects such as resource consumption are out of scope of this paper due to space limitations. From a technical point of view, we require that alternative algorithms share the same input and output (item) types, respectively. Further, we assume that the processing environment is robust, i.e., tuples are processed without failures.

3.1 Simple approach

We introduce now a simple approach for switching among alternative distributed algorithms, which basically relies on re-routing the input stream at runtime. We use this approach as a baseline in our analysis in Section 4.

Figure 1 illustrates the approach in terms of a data flow diagram, i.e., nodes represent (distributable) data processors and edges the data flow. Without loss of generality, we illustrate our approaches using just two alternative algorithms. A setting with multiple alternatives can be constructed similarly. In Figure 1, the processors $P_{1,1}$ to $P_{1,n}$ constitute *Algorithm₁* (akin $P_{2,1}$ to $P_{2,m}$ for *Algorithm₂*), whereby the data flow within the respective algorithm is not relevant for our discussion. In addition, two guarding processors control input (*Switching Element*) and output (*Join Element*) streams. Actually, these guards can be realized as part of proceeding or succeeding processors to save resources.

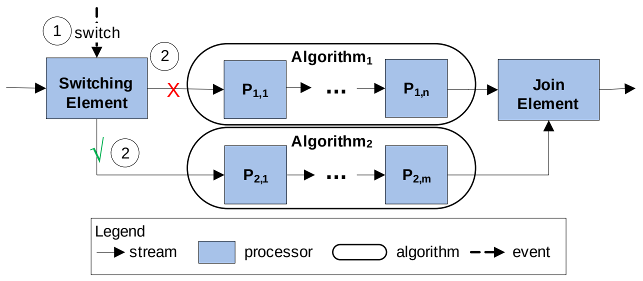

Figure 1: The simple switching approach.

Let us assume that *Algorithm₁* is the currently active algorithm. A signal, i.e., an asynchronous event sent to a processor, indicates the actual need to switch the active algorithm to another target algorithm ①. In the simple approach, the switch signal causes an immediate re-routing of the data streams ②, i.e., the stream to *Algorithm₁* is disabled and the alternative stream to *Algorithm₂* is enabled at the same time. No action is needed in *Join Element*.

Actually, the simple approach does not consider queuing effects in the active algorithm, which can cause the algorithm to continue processing even after the switch. Thus, this approach can significantly increase the switching time and lead to inconsistent results as the processing of both algorithms can overlap in an uncontrolled fashion. In summary, the simple approach likely fails regarding R1 - R4.

3.2 Advanced approach

In this section, we discuss an advanced approach, which aims at minimizing the impact on the underlying data streams through realizing the requirements R1 - R4. The core idea of this approach is to combine four techniques into a single approach, namely:

- **Output stream control** to disable the output stream of the actual algorithm upon switch (R3, R4) and to avoid accidental data overlap (R1).

- **Acknowledgements and buffer transfer** to track and stop processing in the actual algorithm during the switch (R1 - R3). Therefore, we (conceptually) send an acknowledgment signal for each item processed by an algorithm to its input buffer, so that incompletely processed items can be identified and transferred to the target algorithm. As the transfer time depends on the amount of buffered items, we aim at an improved mechanism in future work.

- **State maintenance** to prepare the state of the target algorithm so that it can safely take over processing at a distinct point in time (R3, R4). Currently, we rely on a variant of the parallel track strategy [13], i.e., to run active and target algorithm for a time Δt_m in parallel to enable the target algorithm to create and stabilize its state. Actually, Δt_m depends on the involved algorithms and can, e.g., be the time frame of a sliding window. As mentioned above, relying on a parallel track strategy is due to our project context, where we also consider hardware co-processors.

- **Output synchronization** based on unique item identifiers to ensure the output item sequence and, thus, transparency according to R2. This is important, as the involved algorithms may process data with a different latency.

Combining these four techniques in a distributed processing environment requires the exchange of various types of signals. Figure 2 a) illustrates the design of the advanced approach. Basically, the advanced approach is an extension of the simple approach in Section 3.1, and also includes two alternative algorithms and two guarding processors. For enabling acknowledgment and buffer transfer, we equip both algorithms with an entrance queue. Conceptually, we represent this as two individual processors (*Intermediary₁* and

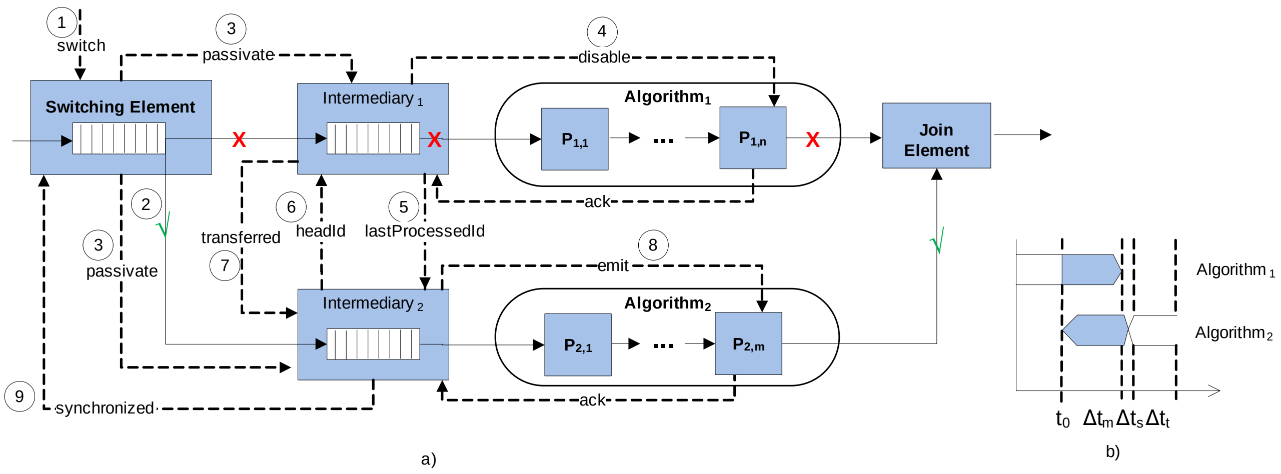

Figure 2: The design of the advanced switching approach in terms of a) signals and b) timing.

Intermediary₂) so that the queues can be maintained independently. A specific implementation may also be based on the queues of $P_{1,1}$ or $P_{2,1}$, respectively. The queues in the intermediary processors remove an item only when it is fully processed by the last processing node of the respective algorithm. This is indicated by an acknowledgment signal (*ack* in Figure 2) sent by $P_{1,n}$ or $P_{2,m}$, i.e., the last node of the respective algorithm. Thus, the items in the queues are either pending to be processed or emitted to the respective algorithm but not fully processed yet. Moreover, we utilize a further queue in the *Switching Element* to control the overall stream ingestion.

Let again *Algorithm₁* be the currently active algorithm and let t_0 be the point in time when *Switching Element* receives the **switch signal** ①. In addition to the overall design depicted in Figure 2 a), we illustrate the individual phases of the approach in the timing diagram in Figure 2 b). The switch signal initiates the runtime algorithm switching. First we **warm-up** *Algorithm₂*, i.e., we run *Algorithm₂* in parallel to build up its state. Therefore, we activate the stream to *Algorithm₂* by duplicating the input items in *Switching Element* ②. At the end, *Join Element* passes only the items of the active algorithm, i.e., it discards the output of the passive algorithms, in particular the output of *Algorithm₂* in the warm-up phase. Now, both algorithms process the input stream in parallel for Δt_m.

The actual switch happens at $t_0+\Delta t_m$ as indicated in Figure 2 b) by performing the **output synchronization**, i.e., *Intermediary₁* negotiates with *Intermediary₂* the last processed item in *Algorithm₁* as a basis for the queue transfer. During switching, each item is queued in the intermediary processors along with a sequential identifier indicating the arrival order of the items. Let *lastProcessedId* be the identifier of the last item emitted to *Algorithm₁*. However, ongoing data processing during the synchronization may invalidate *lastProcessedId*. Therefore, we first passivate ③ both algorithms during the synchronization and disable ④ also the output of results in $P_{1,n}$ to avoid that acknowledgment signals disturb the synchronization. $P_{1,n}$ confirms the passivation (not shown on Figure 2) so that *Intermediary₁* can now send *lastProcessedId* to *Intermediary₂* ⑤. We denote the time needed for synchronization as Δt_s.

Let *headId* be the identifier of the head of the queue in *Intermediary₂*. As the involved algorithms may operate at a different speed, we must consider three cases for the synchronization (illustrated as queues for the intermediary processors in Figure 3):

a) If $headId = lastProcessedId$, both algorithms are running at the same speed. No items must be transferred and *Algorithm₂* can immediately take over the processing from *Algorithm₁* as shown in Figure 3 a).

b) If $headId < lastProcessedId$ then *Algorithm₁* is faster than *Algorithm₂*. In this case, no items must be transferred, but the items [*headId*, *lastProcessedId*] must be skipped as they would cause duplicated results. For example, in Figure 3 b) items [499, 500] have been processed and must to be skipped.

c) If $headId > lastProcessedId$, items (*lastProcessedId*, *headId*) must be transferred to *Intermediary₂*. Therefore, *Intermediary₂* sends the *headId* to *Intermediary₁* ⑥ and initiates the queue transfer, i.e., *Intermediary₁* sends unprocessed items via network to *Intermediary₂*. Let Δt_t be the transfer time. At $t_t = t_0+\Delta t_m+\Delta t_s+\Delta t_t$, *Intermediary₂* is notified about the end of the queue transfer ⑦ to prepare for regular items from *Switching Element*. In the example in Figure 3 c), the lower queue is processed faster than the upper queue. So the items (500, 503) must be transferred to avoid a gap.

Due to the *ack* signals from $P_{2,m}$, *Intermediary₂* can track whether all data items for warm-up have been processed. As soon as synchronized items are passed to *Algorithm₂*, *Intermediary₂* sends an *emit* signal ⑧ to $P_{2,m}$ enabling the

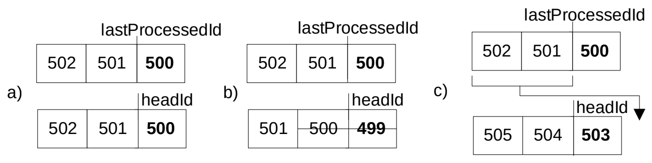

Figure 3: Three cases of output synchronization.

output of processed data to *Join Element*. To minimize the switching time, *Algorithm$_2$* processes synchronized items in parallel to the queue transfer, i.e., it starts processing real data already during queue transfer. Finally, $P_{2,m}$ confirms the activation of the output stream (not shown in Figure 2). In turn, *Intermediary$_2$* notifies *Switching Element* about the end of the synchronization ⑨ as well as that *Algorithm$_2$* took over the processing and *Algorithm$_1$* is discarded.

As discussed, the advanced approach uses a combination of signals, queue transfer, state warm-up and output synchronization to reduce the impact of switching among distributed algorithms at runtime. In this approach, the critical period is Δt_s when synchronization happens and no processing takes place. During queue transfer, *Algorithm$_2$* can already take over processing. Due to the warm-up phase, the overall switching time is at least of $\Delta t_m + \Delta t_s$, but the queue transfer may cause a peak load of items to be processed. However, during the warm-up phase, we utilize the processing resources for two algorithms in parallel. As mentioned already above, optimizing the resource consumption is out of the scope of this paper.

Realizing such a complex signal-based approach requires adequate support for the data analyst, e.g., a framework or code generation as we do in the QualiMaster project.

4. EVALUATION

We present now an evaluation of the approaches discussed in Section 3. The goal of the evaluation is to analyze and validate the actual impact of the proposed approaches with respect to requirements R1 - R4. Therefore, we evaluate in this section both, the simple approach as a baseline as well as the advanced approach. In particular, we focus on the impact on the data streams, i.e., the time for performing an algorithm switch (R3) and, as mentioned above, the throughput as a measure of (R4). In addition, we analyze the realization of R1 and R2. In this section, we discuss first the environment, the implementation of the approaches and the settings we used for conducting the evaluation. Then, in Section 4.1 we evaluate the simple approach and the advanced approach in Section 4.2.

Environment. We conduct our experiments on a Storm (version 0.9.5) cluster composed of one Nimbus machine for managing topologies, 6 worker machines for distributed processing and three Zookeeper instances (version 3.4.6) for managing the workers. Each machine is equipped with an Intel(R) Core$^{(TM)}$2, 1.86GHz CPU and runs Ubuntu 12.04.4 LTS as well as Java JDK version 1.7.0_71. All machines in the cluster are connected over a dedicated network switch via Gigabit-Ethernet at a transfer rate of around 100MByte/s. In addition, we synchronize the system clocks of all machines using the Network Time Protocol (NTP) in order to enable comparisons of the timestamp-based logs written during the experiments. Due to time synchronization, the actual time difference among the machines is less than 3 ms.

Implementation. We implemented both approaches discussed in Section 3 as of Storm topologies consisting of processors called Bolts and data sources called Spouts. The Storm mechanisms for guaranteeing the processing of each item already provide us with an implementation of the acknowledgment signals. We realized the other signals in terms of Zookeeper change notifications, i.e., when data is written by the sender into a dedicated Zookeeper node, the receiver is notified by the Zookeepers about the change and can react on the signal data. All processors discussed in Section 3 except for the intermediary ones are implemented as Bolts. As Bolts work item-wise and Spouts operate asynchronously and their internal queues are not accessible, we realize the intermediary processors in the advanced approach as Spouts with explicit queues. In more detail, we use a dedicated input queue receiving input items and an output queue temporarily storing sent and pending items, removing them upon acknowledgement signals. The connection between the *Switching Element* and the intermediate processors are realized as network connections. For the experiments, a single Spout realizes the stream ingestion and produces sequential integer items at a configurable rate. For achieving repeatable experiments, we use simple distributed algorithms, which consist of a processor that just passes the received input data. To simulate performance differences, the algorithms can be configured with a certain latency (delay) per item. For obtaining experimental data, a processor records the individual arrival time per item in a processor-specific log file. Running such a topology without delays in the algorithms and no switching shows that logging does not significantly influence the nominal throughput.

Setup. In our experiments, we use a constant ingestion frequency of 1000 items per second. *Algorithm$_1$* is the active algorithm at the beginning of each experiment. For both approaches, we perform two experiments, one with delay-free (near-ideal real-time) algorithms and one to analyze the behavior at a certain load. More specifically, in the load situations *Algorithm$_1$* acts as a slow algorithm with a delay of 30ms per item. Based on the window time of a specific scenario in our project context, we use 30s as warm-up time in this evaluation. The experiments are implemented as Java programs managing the topologies and issuing the switch signal so that the experiments become repeatable.

We discuss now the results of evaluating the simple and the advanced approach. For each experiment, we repeatedly run the respective experiment implementation at least five times in order to detect deviations. While we identified and fixed deviations in pre-experiments, there were no significant deviations in the final experiments, so that we report results of typical executions in this paper. Although the log files written during the experiments are recoded in terms of milliseconds and the analyses were done based on these logs, we illustrate below the results by overview figures in terms of seconds time unit.

4.1 Simple approach

In this section, we discuss the evaluation of the simple approach presented in Section 3.1. As the best case, both algorithm runs without delay, i.e., they do not queue data. In this case, our experiments show no negative impact on the throughput. However, the more interesting case includes an algorithm that processes at a certain latency so that queuing effects can occur.

Figure 4 illustrates the throughput, i.e., the number of data items per second logged in the *Join Element*. In the beginning, the data processing is running on the slower *Algorithm$_1$*. At ①, we send the switch signal and cause a rerouting of the data streams from *Algorithm$_1$* to *Algorithm$_2$*. As depicted in Figure 4, the throughput after the switch fluctuates for around 15 seconds. To explain this effect, we analyzed the logs written by our implementation of the processors. By comparing the timestamps of the items, we re-

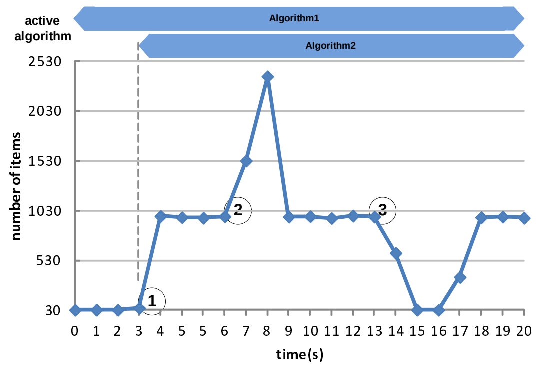

Figure 4: Throughput caused by the simple approach when switching from a slower algorithm.

alized that the increasing throughput at ② is due to the queuing of items in the preceding *Switching Element* caused by the slow algorithm. Instead of coping with the queued items immediately, in this experiment, the processing of the queued items is deferred by 3 seconds. Further, the reduced throughput at ③ is caused by the queuing of items in the (internal queues of) $Algorithm_1$ due to its configured latency. The queuing in $Algorithm_1$ keeps its processors still active as indicated by the timing bars in Figure 4 although the data stream is already processed by $Algorithm_2$. Thus, the simple approach significantly increases the switching time until all items queued in the $Algorithm_1$ are processed. In this experiment, this leads to inconsistent results as both algorithms overlap processing after the switch. The logs of $Algorithm_1$ and $Algorithm_2$ indicate that 1427 out-of-order items are caused by the ongoing processing in $Algorithm_1$, which affects the overall output accuracy. As a result, we conclude that the simple approach does not fulfill R1 - R4.

4.2 Advanced approach

In this section, we evaluate the advanced approach presented in Section 3.2 in terms of both, switching among the same speed algorithms as well as from a slower algorithm.

Figure 5 depicts the throughput while switching among algorithms operating at no delay. After the actual switch signal arrives at ①, both algorithms run in parallel for the warm-up phase until ②. As items are processed by both algorithms at the same speed, no items are transferred (the first case in Section 3.2) as indicated by the experiment logs.

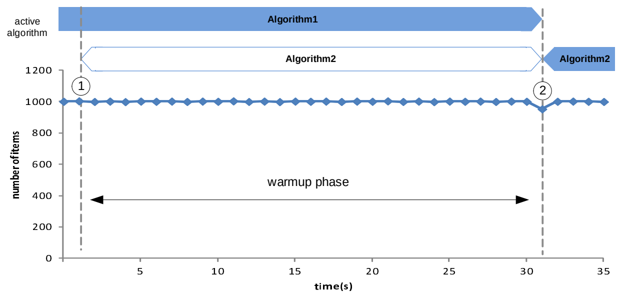

Figure 5: Throughput of the advanced when switching between algorithms at the same speed.

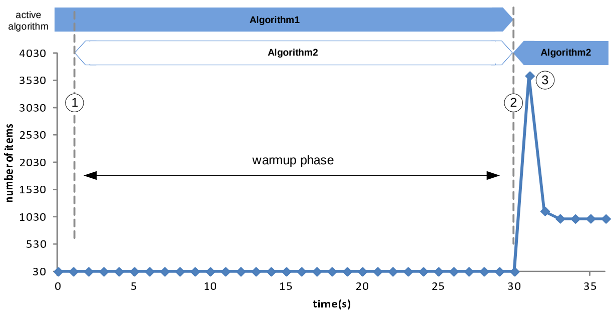

Figure 6: Throughput of the advanced when switching from a slower algorithm.

At ②, $Algorithm_1$ is terminated and $Algorithm_2$ takes over the processing. Based on the experiment logs, we also found that actually no items are synchronized. Further, on the output steam neither items are missing nor duplicated (R1). Moreover, the output item sequence corresponds first to the sequence produced by $Algorithm_1$ and, after the switch, by $Algorithm_2$ (R2). The actual impact is a short reduction of the throughput to 950 items, which is caused by the output synchronization. In this experiment, we identified the actual switching time as 60ms from the logs. This is reasonable as 6 signals are exchanged and in our cluster the time between sending and receiving an event is in average around 10 ms.

Figure 6 illustrates the throughput while switching from a slower algorithm at 30 ms delay to a faster one. The actual switch signal arrives at ① and again both algorithms run in parallel until ②, the end of the warm-up phase of 30s. Please note that during the warm-up phase the output is still produced by the slower $Algorithm_1$. The output synchronization starts at ②. Switching from a slower algorithm, i.e., the third synchronization case discussed in Section 3.2, leads to a queue transfer from $Algorithm_1$ to $Algorithm_2$. In this experiment, 2889 items are transferred and the total transfer takes Δt_t=820 ms as indicated by the experiment logs. As discussed in Section 3.2, $Algorithm_2$ starts processing data in parallel to the queue transfer so that the switching time is not dominated by Δt_t. However, several transferred items arrive in short time at $Algorithm_2$ and lead to a throughput peak ③. As we analyzed from the logs, in this experiment the actual switching time takes 106 ms, mostly due to the 9 signals (each taking around 10 ms) sent during the switch.

5. CONCLUSIONS AND FUTURE WORK

Distributed data stream processing is a popular approach to realize Big Data Applications. While processing data streams, the actual stream characteristics may change dramatically, e.g., when sentiments change in the Social web or when stock markets become hectic. To cope with such changes during processing and to provide a steady output quality, data processing must adapt to the actual context.

In this paper, we discussed the runtime switching among (distributed) data processing algorithms as one specific form of realizing adaptive stream processing. We introduced an advanced switching approach, which takes queuing effects into account, maintains the output sequence, and, to reduce the switching time, utilizes parallel track processing to warm-up the target algorithm. We implemented both approaches on Apache Storm, performed experiments and analyzed the results. As expected, plain stream re-routing suffers from queuing effects, item duplication and requires an overall switching time of more than 15 s. In contrast, on our cluster, the advanced approach reduces the switching time to 60 ms (algorithms with no latency) or less then 110 ms (at 30 ms latency) without disturbing the output sequence. While effects regarding timeliness occur due to the processing of queued items, we showed the effectiveness of our approach in reducing the impact on the data streams.

Currently, we integrate the advanced approach into the model-based generation of topologies developed in the QualiMaster project to unburden the data analyst from manually implementing efficient adaptation that require queuing and complex signal interaction. Ultimately, this will include the generation of transparent integrations of hardware co-processors. Here, initial results in combination with dynamic switching among algorithms are promising.

In the future, we plan to improve our approach, in particular to speed up the events, to limit the queue transfer time and to research state transfer mechanisms that do not need a parallel warm-up, in particular for switching between software-based processing and hardware co-processors. Furthermore, we aim at taking further stream characteristics into account as well as improving the overall resource usage, e.g., by allocating resources only for active algorithms and, if needed, during the warm-up phase. Moreover, we will consider the impact of switching on the result quality through quality measures and consider how processing failues can be handled to guarantee robustness.

6. ACKNOWLEDGMENTS

This work was partially supported by the European Commission in the 7th framework programme through the QualiMaster project (grant 619525).

7. REFERENCES

[1] A. M. Aly, W. G. Aref, M. Ouzzani, and H. M. Mahmoud. JISC: Adaptive Stream Processing Using Just-In-Time State Completion. In *International Conference on Extending Database Technology, (EDBT' 14)*, pages 73–84, 2014.

[2] H. C. M. Andrade, B. Gedik, and D. S. Turaga. *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press, 2014.

[3] C. Balkesen, N. Tatbul, and T. M. Özsu. Adaptive Input Admission and Management for Parallel Stream Processing. In *International Conference on Distributed Event-based Systems (DEBS '13)*, pages 15–26, 2013.

[4] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous Dataflow Processing. In *International Conference on Management of Data (SIGMOD '03)*, pages 668–668, 2003.

[5] H. Eichelberger, C. Qin, K. Schmid, and C. Niederée. Adaptive Application Performance Management for Big Data Stream Processing. In *Symposium on Software Performance (SSP '15)*, 2015.

[6] J.-H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *International Conference on Data Engineering (ICDE '05)*, pages 779–790, 2005.

[7] Y. Ji, H. Zhou, Z. Jerzak, A. Nica, G. Hackenbroich, and C. Fetzer. Quality-Driven Continuous Query Execution over Out-of-Order Data Streams. In *International Conference on Management of Data (SIGMOD '15)*, pages 889–894, 2015.

[8] D. Kulkarni, C. V. Ravishankar, and M. Cherniack. Real-time, Load-adaptive Processing of Continuous Queries over Data Streams. In *International Conference on Distributed Event-based Systems (DEBS '08)*, pages 277–288, 2008.

[9] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: Stream Processing at Scale. In *International Conference on Management of Data (SIGMOD '15)*, pages 239–250, 2015.

[10] Q. Lin, B. C. Ooi, Z. Wang, and C. Yu. Scalable Distributed Stream Join Processing. In *International Conference on Management of Data (SIGMOD '15)*, pages 811–825, 2015.

[11] B. Liu, Y. Zhu, M. Jbantova, B. Momberger, and E. A. Rundensteiner. A Dynamically Adaptive Distributed System for Processing Complex Continuous Queries. In *International Conference on Very Large Data Bases (VLDB '05)*, pages 1338–1341, 2005.

[12] K. Rothermel and T. Helbig. An adaptive protocol for synchronizing media streams. *Multimedia Systems*, 5(5):324–336, 1997.

[13] E. A. Rundensteiner, L. Ding, Y. Zhu, T. Sutherland, and B. Pielech. CAPE: A Constraint-Aware Adaptive Stream Processing Engine. In N. A. Chaudhry, K. Shaw, and M. Abdelguerfi, editors, *Stream Data Management*, pages 83–111. Springer, 2005.

[14] M. Shah, J. M. Hellerstein, S. Chandrasekaran, M. J. Franklin, et al. Flux: An adaptive partitioning operator for continuous query systems. In *International Conference on Data Engineering (ICDE '03)*, pages 25–36, 2003.

[15] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47, 2005.

[16] R. Tudoran, O. Nano, I. Santos, A. Costan, H. Soncu, L. Bougé, and G. Antoniu. JetStream: Enabling High Performance Event Streaming Across Cloud Data-centers. In *International Conference on Distributed Event-Based Systems (DEBS '14)*, pages 23–34, 2014.

[17] Y. Wei, S. H. Son, and J. A. Stankovic. RTSTREAM: real-time query processing for data streams. In *International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC '06)*, pages 141–150, April 2006.

[18] S. Wu, V. Kumar, K.-L. Wu, and B. C. Ooi. Parallelizing Stateful Operators in a Distributed Stream Processing System: How, Should You and How Much? In *International Conference on Distributed Event-Based Systems (DEBS '12)*, pages 278–289, 2012.